SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract. **10**, 93–110 (1998)

Practice

Using Application Understanding to Support Impact Analysis

M. JOANNA FYSON and CORNELIA BOLDYREFF*

Centre for Software Maintenance, Department of Computer Science, University of Durham, Durham DH1 3LE, U.K.

SUMMARY

Any change to one component of a software system is likely to be felt in other components, a phenomenon known as 'ripple effect'. Impact analysis is the process of identifying all those components of a system which may be affected by ripple effect. This requires a knowledge of the dependencies between components.

A software system consists not only of the source code, but also of all life cycle work products including requirements, design and test documents. Any of these components may be the subject of a change request, and should therefore be included in impact analysis.

The ripple propagation graph (RPG) for impact analysis, developed as part of the AMES (Application Management Environments and Support) project, models an application in such a way that dependencies can be traced in order to identify all affected components. It is an object-relationship model consisting of the components of a system at some level of granularity, and the various relationships between them.

This paper describes the AMES RPG, and how the results of the application understanding (AU) toolset, developed (again as part of the AMES project) to support the process of gaining an understanding of an application, can be used to provide some of the data required to populate the RPG. © **1998 John Wiley & Sons, Ltd.**

KEY WORDS: application understanding; impact analysis; ripple propagation graph; ripple effect; application management support; traceability

1. INTRODUCTION

With the ever-growing cost of development of software systems it becomes increasingly necessary to extend the life of existing systems, adding new functionalities as required or adapting them to new environments, a process known as maintenance. Before a change can be implemented, a detailed knowledge of the application must be acquired in order to determine where and how to make the required change. The process of gaining this

* Correspondence to: Cornelia Boldyreff, Centre for Software Maintenance, Department of Computer Science, University of Durham, Durham DH1 3LE, U.K. E-mail: Cornelia.Boldyreff@durham.ac.uk

Contract/grant sponsor: European Union; Contract/grant number: 8156 AMES

CCC 1040–550X/98/020093–18$17.50
© 1998 John Wiley & Sons, Ltd.
Received 19 September 1996
Revised 20 December 1997

knowledge is termed application understanding (AU). Where an application is viewed as consisting of all life cycle documentation associated with its development and maintenance—requirements specification, design documents, test cases, technical documentation, etc.—in addition to source code, a change may initially be made to any one of these components, and corresponding changes must then be made to related documents. Thus, a decision to add some new functionality may initially be specified in the requirements specification, but would result in modifications to design documents and ultimately to the code. Conversely, any changes made at code level, e.g., changing the type of a data structure, should be recorded in the relevant design documents. Thus, it can be seen that the need for AU extends beyond the low level program comprehension required to identify those source code units to be changed to an understanding of the relationship between components at a high level of abstraction.

Whenever a change is made to a software system there is the risk of introducing new errors, one cause of which is the failure to make corresponding changes to related components (Agusa, 1983). A change to one component of a software system is likely to affect other components, and the resulting changes in those components will similarly and iteratively affect further components. This 'ripple effect' can cause the effect of the initial change to be propagated far from its starting point. In order to perform successful maintenance, therefore, it is necessary to identify *all* such affected components, a process known as impact analysis (Arthur 1988).

Impact analysis requires a knowledge of the dependencies between components. Dependent on the nature of the task in hand, 'components' in impact analysis may refer at one extreme to functions, data structures, etc. in source code, or at the other to complete documents, such as the requirements specification or a design document. Similarly, dependencies may range from calls between functions or uses of a variable, to the relationship between a requirements definition and the design document which refines it. Thus, if our task involves changing the type of a parameter to a function, impact analysis must be undertaken at a detailed level within source code in order to identify other functions which use that data item. However, in order to maintain consistency within all system documentation, impact analysis is also required at a high level to determine the particular design document relating to those functions. Thus, it can be seen that there is a requirement for impact analysis to be undertaken at various levels of abstraction, both within a life cycle stage and across life cycle stages.

In addition, it is necessary to be able to trace dependencies both forwards (e.g., from a requirement to the relevant design and code modules, or from a calling function to a called function), and backwards (from a source code element to the relevant design, or from a function to its calling functions). Finally, in a well-managed maintenance environment there is the need to perform impact analysis of a proposed change *before* implementation commences in order to gather information to support decision-making. Such decisions involve comparison of possible solutions to determine the risks involved, and also the related costs.

It has long been established that application understanding and impact analysis are difficult and time-consuming, and therefore costly, tasks, accounting for a large proportion of total maintenance costs (Lividas and Alden, 1993). However, there are few supporting tools. Those that do exist concentrate on source code, and impact analysis tools assist the

© 1998 John Wiley & Sons, Ltd.

maintainer only to identify affected modules *following* some change. There is thus a need for automated support, both for initial understanding acquisition to identify components and their inter-relationships, and to retain such information once discovered for future maintenance interventions.

The AMES project is dedicated to developing a total maintenance support environment. The importance of application understanding and impact analysis are recognized, and the requirement for tool support identified. Therefore, AU Toolset has been developed, consisting of tools which can analyse a variety of forms of software documents, resulting in a united graph (UG) which maps all related aspects of the source code and documentation. The UG allows the maintainer quickly to trace a concept (key word or phrase) through a document, and to follow links between different life cycle representations of some entity within requirements, design and code documents in order to further his/her understanding of that entity.

To support impact analysis, a further tool (the Impact Analysis System, or IAS), has been developed which can semi-automatically analyse an object-link graph in order to identify objects which are related in some way to a specified object (Barros *et al.*, 1995). A generic model of software systems, the ripple propagation graph (RPG) has been specified, such that objects in the RPG represent generic components of systems at some level of granularity, and links represent the various dependencies that may exist between them. Given a set of generic rules concerning how changes of a particular nature may propagate to other components in the system, the IAS can identify those components which will (or may be) affected by a specified change.

The problem remains of relating these generic concepts to a particular application. The generic RPG (or part of it) must be instantiated with the actual objects and dependencies which exist in the application in question. The discerning reader will no doubt already have surmised that the understanding gained through use of the AU Toolset can be utilized for this purpose. The focus of this paper is to describe how the output of the AU Toolset can be used to generate automatically the instantiated model (a dependency file) which can then be fed directly into the IAS in order to perform impact analysis. The work so far completed allows impact analysis to be undertaken at a high level of abstraction, thus providing support for the pre-maintenance decision-making process.

The remainder of this paper first looks at the background to this work, identifying some of the problems of the maintenance task and describing the AMES approach to solving it, and surveying earlier work in the areas of application understanding and application modelling. It then describes the AU Toolset and AMES RPG in greater detail, and how the output of the AU Toolset can be utilized to populate the RPG for impact analysis at a high level of abstraction.

2. BACKGROUND

2.1. Maintenance—some problem areas

Maintenance refers to the activities that take place after a software product has been delivered to the customer. Investigations into the types of changes made during the process

© 1998 John Wiley & Sons, Ltd.

of maintenance by Lientz and Swanson, among others (Lientz, Swanson and Tompkins, 1978; Swanson, 1976), have led researchers to classify software maintenance into a number of different categories. A commonly used classification is as follows:

- adaptive maintenance—modification of a software product performed to account for changes in the hardware or software operating environment;
- corrective maintenance—reactive modification of a software product to correct discovered faults;
- perfective maintenance—modification of a software product to meet new or changing user requirements; and
- preventive maintenance—changes made to an application with the purpose of improving the future maintainability of the software.

Whilst each of these types of maintenance carries its individual problems, many problems are common to all. Predominant among these are the following:

- The fact that in the past much software was produced by *ad hoc* software engineering methods, which means that formal specification and design documentation etc. are often non-existent; without the more abstract views of a system provided by such documents, the task of understanding an application is that much more difficult.
- Past maintenance projects have often compounded the problems of poor documentation by making this already inadequate documentation also out of date.
- Poorly structured code frequently means that it is difficult to investigate the potential effects of changes becaues of the ripple effect, therefore performing pre-maintenance cost analysis is practically an impossible task.
- Tool support for the maintenance process is limited and generally inadequate.

The result of these problems is an ever-increasing cost of the maintenance function. Pressman (1992) reports that in the 1970s maintenance costs were 35–40% of total software life cycle costs, jumping to approximately 60% in the 1980s. Nowadays this can be anything up to 80% of total costs.

The AMES project was initiated to attempt to alleviate some of these problems. The next section briefly describes the aims of the project, focusing on the aspects of application understanding and impact analysis.

2.2. The AMES project

AMES (Application Management Environments and Support) is an ESPRIT project whose aim is to provide methods and tools for an effective support to application management. Application management is the contracted responsibility for the management and execution of all activities related to the maintenance and evolution of existing applications (AMES Technical Annex quoted in Boldyreff, Burd and Hather (1994)).

AMES provides a complete methodological framework for the maintenance process and a set of tools to support the most critical activities of application management. These tools are built on top of a traceability platform (TP), which provides a pool of information

© 1998 John Wiley & Sons, Ltd.

about an application in the form of objects, being entities within the application, and relationships between those entities. A set of navigation and display tools offers the user search facilities to enable rapid retrieval of components in the TP, together with graphical displayers which offer several views of objects and links contained in the traceability base.

As mentioned earlier, one of the problems facing maintainers is the lack of life cycle documentation. One set of AMES tools has been developed to support reverse engineering of C programs into well-structured and well-documented HOOD designs (Battaglia and Savoia, 1995). In addition, Cobol analysers have been included to capture information from Cobol programs and allow the maintainer to improve his view of what is in a typical commercial application.

The remaining groups of tools, for application understanding and impact analysis, have already been mentioned and will be described in greater detail in subsequent sections of this paper.

2.3. Background theory for the AU Toolset

Gaining an understanding of the detailed workings of an application is an essential prerequisite for the maintenance process. Obviously, before a change can be implemented a maintainer must be able to identify areas of the code requiring modification to achieve that change. In addition, in a well-managed maintenance environment maintainers are expected to be able to provide appropriate and accurate assessments of the maintenance service to be provided. They therefore need to be able to estimate the impact on a program and its supporting documentation that a proposed change will entail. This requires a high degree of program comprehension (understanding) about the application to be maintained, and this must be gained in advance of any actual maintenance. This is both an expensive and lengthy process.

The AMES AU Toolset has been developed with the aim of improving the efficiency of the maintenance process by providing support for the maintenance engineer in his task of understanding a system. Its requirements were determined from a study of the theory of learning and of empirical studies of program comprehension in maintenance (Burd, Younger and Boldyreff, 1994). These are summarized below.

2.3.1. The theory of learning

Study of the theory of learning has revealed that important factors in facilitating the understanding process are the structure and presentation of information, and the sequence in which it is provided; and that there is also a distinct phased process of learning which is both iterative and incremental (Boldyreff *et al.*, 1995).

Structure refers to the hierarchical ordering of material. For effective learning, the amount of information presented should be limited, and it should be presented in such a way as to enable the association of separate items of related data.

Information should be presented in its most suitable form, and this varies according to the type of information and the needs of the maintainer. A graphical view enables some types of information to be presented more succinctly, clearly and accurately, and to be mapped more readily onto our mental representation of information. A textual view

© 1998 John Wiley & Sons, Ltd.

provides an alternative view to the graphical display and is more appropriate for some types of information. In addition, much information is already represented in this form.

Finally, sequence refers to the order in which information is presented. Understanding of complex problems is greatly helped by appropriate sequencing of information. This allows a complex problem to be reduced to a series of simpler understanding tasks.

Gagne (1974) suggests that the process of learning consists of eight phases (Figure 1), where each phase describes a continuous process whereby information within one instructional block is selectively collected, stored and retrieved. The overall process of learning is represented by numerous iterations of this phased approach. Each learning phase defines an 'instructional event' that indicates how the learning process may be aided. It is these which the AU Toolset seeks to support.

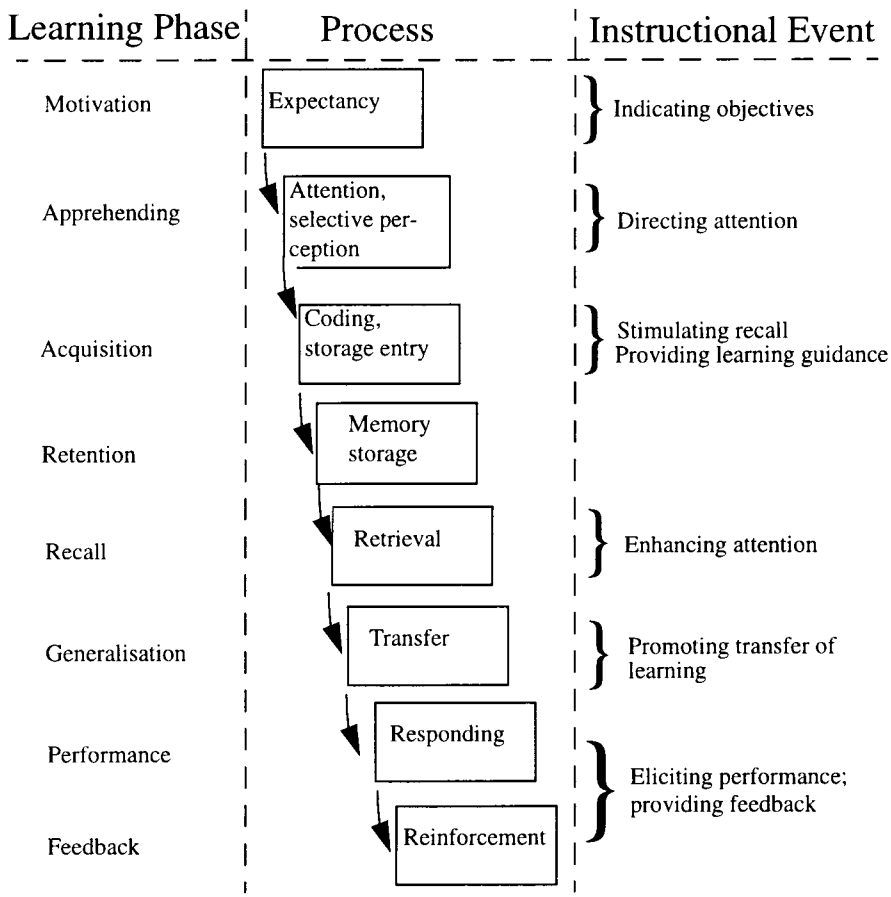

Figure 1. Relation of phases of learning to instructional event (taken from Gagne (1974))

© 1998 John Wiley & Sons, Ltd.

2.3.2. Empirical studies of maintenance

Several studies into the process of program understanding have been undertaken (Younger and Bennett, 1993) resulting in a number of theories as to how understanding is gained. Brooks (1983) suggests that comprehension is based on a system of mapping between the problem domain and the programming domain. The maintainer's task consists of reconstructing the mappings made by the original developer, using whatever information is available. This may be achieved in a bottom–up or top–down manner. Letovsky (1986) argues that comprehension typically requires a mixture of both.

The bottom–up approach begins by analysing the syntax of program statements in order to derive semantic information, hence identifying plans and algorithms in the code. These then form a basis for reasoning about the high-level functions of the program. The top–down method is an iterative process of hypothesis refinement and validation (Gilmore, 1991). The programmer begins with an initial hypothesis about the functionality of a program, routine or module based, for example, on its name or an application domain knowledge. Software engineering knowledge will suggest ways in which this functionality may be implemented, i.e., which plans are to be expected. The programmer then tests this initial hypothesis by examining the code, attempting to identify the statements which implement the expected high-level concepts, and iteratively refines both the initial hypothesis and his mental model of the actual program until they match (see Figure 2).

A further facet to the manner in which a maintainer seeks to understand a program concerns the proportion of the program studied. Taking a systematic approach, the maintainer studies and attempts to understand the entire program before beginning to modify it. Alternatively, the maintainer may adopt an as-needed strategy where he studies and understands only those parts of the program which are deemed necessary to carry out the particular task. To implement the as-needed approach the maintainer must first identify the parts of the application which are relevant to his/her higher task, and this is generally achieved by developing a complete understanding of the application at a high level of abstraction in a top–down fashion.

2.3.3. Using the theory of learning to support application understanding

Application understanding is primarily a learning exercise. The model of program comprehension shown in Figure 2 can be taken to characterize the broader concept of application understanding. This model can be broken down into two separate learning processes, represented by the left- and right-hand sides of the diagram, which influence each other through the *match* and *evaluate* processes.

As explained in Burd, Younger and Boldyreff (1994), the effect of this influence is to cause repeated iterations through (parts of) the learning process of Figure 1. The code and documentation on the one hand, and the problem statement on the other, form the material on which an understanding must be developed, guided by the *objective* that the understanding or mental representation of each must ultimately match. The various forms of prior knowledge—software engineering knowledge, domain knowledge, etc. serve to *direct the attention* of the maintainer, and the match and evaluate processes serve either to *reinforce* the existing mental models or to *direct attention* during the next iteration

© 1998 John Wiley & Sons, Ltd.

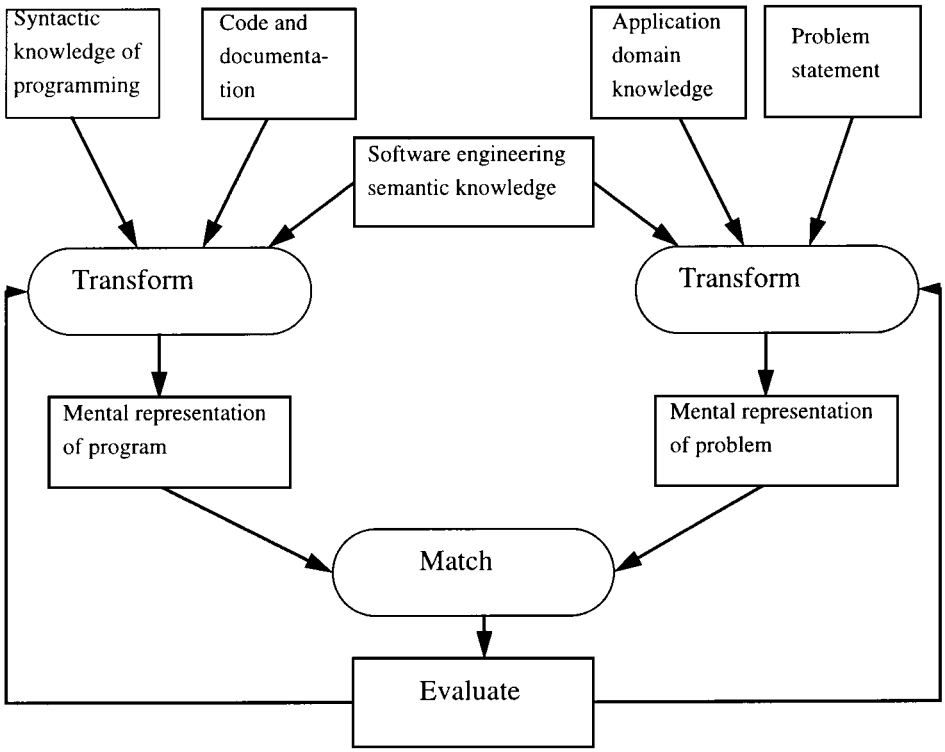

Figure 2. A model for program comprehension (taken from Younger and Bennett (1993))

through the learning process. The *transform* processes of Figure 2 correspond to the middle phases of Figure 1, specifically the *acquisition*, *retention* and *recall* phases.

2.4. Foundations for the AMES RPG

AMES recognizes the fact that impact analysis should be conducted early in the maintenance process in order to provide a measure of control over the process. Management needs to be able to evaluate, in advance, the cost of performing a change, and to compare alternative solutions to determine the best means of accomplishing a required change. This requires identification of all system components that will be affected, at a higher level of abstraction than existing code analysis techniques permit. Turver (1993) proposes the use of a ripple propagation graph representing both the hierarchical and thematic structure of documentation in a software system, as the basis of impact analysis, and this forms the basis for the AMES RPG with refinements and extensions derived from work by Fillon (1994). This section summarizes these two areas of work.

© 1998 John Wiley & Sons, Ltd.

2.4.1. Turver's RPG

Turver's work addresses the problem of tracing the ripple effect of a change at a stage earlier in the maintenance process than existing methods allow. He identifies requirements and design specifications as the most useful in detecting the impact of a change.

His graph is based on a model of documentation consisting of documentation entities, volume entities, chapter entities, section entities, subsection entities and segment entities, where a segment is defined as a standard sized unit of documentation which can be characterized in terms of document types. It concentrates on describing the processing of data at the segment level, and the dependencies between these segments. This necessitates a representation of the relevant content of documentation. Turver achieves this through the use of *themes*, where a theme consists of a conceptual object (a noun) and a conceptual action (a verb indicating some action on the conceptual object). Example themes are a data flow value assignment or the call of a nested procedure. These themes can be considered as categories in order to enable the classification of document segments, which then facilitates impact analysis at the documentation level.

Turver's RPG consists of a hierarchical interconnection graph which provides a picture of the composition and structure of a document; a thematic interconnection graph showing the thematic structure of documentation; a source code association graph providing information regarding those source code components of a system which are associated with particular document entities within a system; and a weighted interconnection graph providing information regarding the likelihood of an indirect impact between two entities by adding weighted edges between segment entities to indicate the strength of the relationship between them.

2.4.2. Fillon's impact analysis for design

Turver's RPG concentrates on the representation and analysis of high level documentation to aid impact analysis at an early stage in the maintenance process. Links to source code modules described in a documentation entity facilitate identification of source code which will be affected by an impact, but there is no support for identifying the effect of a proposed change on design documentation which is in a form other than simple text.

Fillon has developed an interconnection model which traces dependencies between design artefacts, based on a representation of HOOD designs as an entity relationship model. This consists of objects, operations, interfaces, etc. identified by analysis of HOOD object descriptions, and relationships which support design principles described by the HOOD method including the *use*, *include* and *implemented_by* relations. Attributes are used to *qualify* entities, and additional links are introduced to support propagations.

3. THE AMES APPLICATION UNDERSTANDING TOOLSET

It has long been established that a large proportion of maintenance costs can be attributed to the cost of gaining an understanding of an application. Some studies have concluded that this can be at least 50% of total maintenance effort (Standish, 1984).

© 1998 John Wiley & Sons, Ltd.

There exists a number of techniques for the automatic analysis of software programs, but effective maintenance requires an understanding of the functioning of an entire application in its working environment. This involves analysis of all system documentation, not just source code. The AMES AU Toolset produces different views of an application through analysis of all available documentation, and integrates them into a combined multi-level view, the united graph.

This section first explains how the theory of learning has been utilized in the design of the AU Toolset, before describing the AU Toolset itself.

3.1. Structure

The AU Toolset uses the concept of traceability to provide information with a suitable structure to support application understanding. This allows the linking of related information so that a particular idea or theme can be traced through the available documentation. The AU Toolset generates such links, and uses hypertext for its output, allowing such links to be followed using a suitable hypertext browser.

Two types of traceability are supported. Vertical traceability represents traces between associated modules of different stages of the life cycle. For example, requirements have links to the design modules which implement them. Vertical traceability assists program comprehension by allowing maintainers to gain a higher level understanding of an application. Horizontal traceability is the trace of related data across a single life cycle stage. For example, it might indicate the passing of a message through aspects of an object-orientated design, or that a module uses another module. Horizontal traceability thus indicates the direction of the flow of information, and thence the impact of a proposed change at a particular level of abstraction.

Traceability may also be forwards or backwards, referring to the direction in which relationships are explored. Forwards traceability allows one to move from a given entity to related entities, whilst backwards traceability allows one to return from those related entities to the start point.

3.2. Presentation

The AU tool enables information to be viewed in both graphical and textual formats. The graphical representation is an object-relationship graph where the objects are components of the application and the links represent various relationships between them. The graphical presentation tool allows the collapsing of a selected subgraph to a single node, the viewing or hiding of certain types of nodes or edges, and slicing for aspects of the application and viewing only nodes and edges in that slice.

The AU Toolset enables the construction of call graphs and control flow graphs. A call graph represents a program as a group of functions and models the control flow between these functions. The nodes of this graph represent the functions of the program and the arcs represent function invocation. A control flow graph represents the sequence of execution in a program unit, in which nodes represent branching points or subprogram calls in a program, and arcs represent linear sequences of code. From the control flow graph an analysis can show the structure of the program, starts and ends of program

© 1998 John Wiley & Sons, Ltd.

segments, unreachable code and dynamic halts, branches from within loops, entry and exit points for loops, and paths through the programs.

3.3. Sequence

Where a large amount of documentation exists for an application, the maintainer needs to be able to focus on a small part. The AU Toolset uses the technique of graph slicing to restrict the available information to a chosen part. The graph slicing is achieved by manipulating the graphs as follows: specified nodes may be removed or kept, leaf nodes may be removed, specified subgraph nodes may be removed, e.g., all 'called' nodes, nodes with specified levels of fan-in and fan-out may also be removed. The success of the slicing mechanism is dependent upon the type of data available in the graph. The more comprehensive the range of maintenance information and link types, the more useful is the slicing process. Therefore the output of the AU Toolset is a 'united graph', combining the results of the various forms of document and code analysis undertaken by the constituent tools.

3.4. The AU Toolset

The AMES AU Toolset uses all available documentation to support the maintenance engineer in his task of understanding a system. It consists of tools to perform linguistic, code and life cycle analyses. Linguistic analysis can be performed on any textual document available for the system. The *wordlist* utility scans a document and produces a frequency list of the words used within it. This information allows the maintainer to decide which combinations of key words may be best employed when performing the next stage of document analysis. The *link-concepts* utility generates a series of links between given key words/phrases in order to produce horizontal traceability within the document.

The code analysis tools allow a maintainer to produce a graph representation of a program (currently only C programs). The utilities *gen-cg-c* and *gen-cfg-c* generate call graphs and control flow graphs, respectively. These graphs can then be manipulated using a set of graph manipulation utilities to provide focused views of an application. For example, the utility *rmfanin* removes all nodes from a graph which are called more than a specified number of times, so eliminating, e.g., library routines.

The *InName* tool developed by VTT (Laitinen, 1995) is an additional code analysis tool which attempts to replace abbreviated names within code with 'natural' names. These are names which are constructed from whole words of a language using the grammatical rules of that language, thus being more meaningful than abbreviated names and so aiding comprehension. A first prototype of a further tool to support the creation of natural names has been developed for possible future inclusion in the toolset. The natural naming repository tool (Fyson, 1995) aims to enable a software or maintenance engineer to create an on-line repository of potential natural names for objects within an application, using the method of disciplined natural naming (Laitinen and Mukari, 1992).

The life cycle analysis tools generate links between documents produced at different stages of the life cycle. The *extract_req* tool takes an html document and produces a list of the requirement specification numbers based on heading identification. (This tool is

© 1998 John Wiley & Sons, Ltd.

currently specific to documentation produced to ESA standards.) The resultant file is used as input to the *make-lifecycle-graph* tool which generates links between a design document and a requirements document, and between a design document and source code.

The various tools within the toolset are accessed through a TCL/TK graphical user interface which more readily allows the maintainer to pass the necessary parameters and file names to a desired utility.

4. THE AMES RPG

The primary purpose of impact analysis is to identify those entities within an application which will be impacted as a result of some proposed change. In order to ensure that all impacts are identified, it is necessary to be able to trace all relationships between components, both those within an individual life cycle stage and those which cross life cycle stages, and also those to other related entities, such as supporting documentation.

A secondary aim of impact analysis is to enable an assessment of the cost of a change to be made at an early stage, or to enable comparison of alternative solutions. This necessitates the recording of information regarding costs of making changes, and the probabilities of a change being propagated to other areas.

The AMES RPG combines all such information within a single model. The basis of the model is an object-relationship graph, where the objects represent the components of an application, and the links indicate the various relationships between them. It is in fact the union of several graphs (see Figure 3), each representing a particular view of an application.

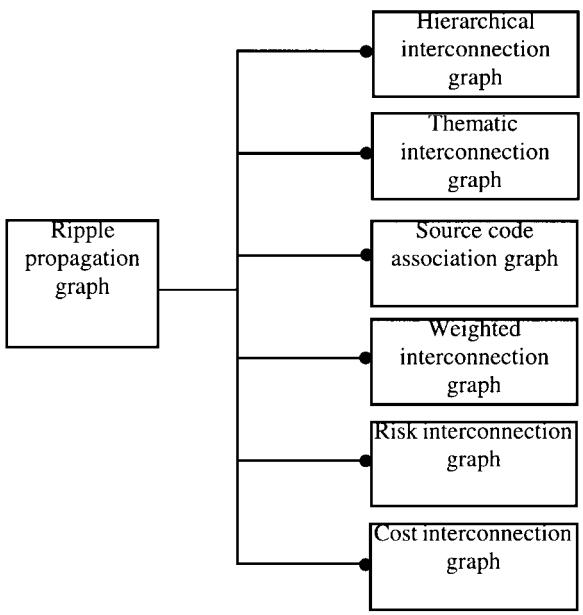

Figure 3. The AMES RPG

© 1998 John Wiley & Sons, Ltd.

Based on Turver's RPG, the AMES RPG consists of a number of subgraphs, as described below.

4.1. Hierarchical interconnection graph (HIG)

A software application consists of many documents, each of which possesses some topology which can be used to support the impact analysis process. For example, a typical document will contain a table of contents, chapters, paragraphs, index, etc., each of which contains elements at a lower level of granularity, e.g., the table of contents is composed of its individual items.

The HIG provides a picture of the composition and structure of a document. Document entities are decomposed into segment entities, these being the smallest entity, and each entity in the hierarchy is allocated an entity type. *Consists_of* dependencies and *entity_type* dependencies show the composition of a document and the role of each entity.

4.2. Thematic interconnection graph (TIG)

The TIG shows the semantic links between elements of an application within one phase of the software life cycle by representing segment entities according to their thematic content. The precise specification of a TIG depends on the particular life cycle stage and the language used, as described below:

- Requirements documents: thematic links in a requirements document enable a given word or phrase to be traced through the document, or link an index entry to its occurrences within the document (the *groups* link). Design documents thematic links at the design level are based on Fillon's specification. In an object-orientated design these include *provides_IF* links between an object design and its interface design, and *defines* links between an object and its operations.
- Implementation documents: TIGs for source code are dependent on the programming language used. TIGs have initially been specified for C and for Cobol source. Objects in the TIG for a C source program include file, type, function, constant and variable. Links include *contains*, *calls*, *reads*, *is_of_type*, etc. The TIG of source code might, for example, be a control flow graph or a call graph.

4.3. Source code association graph (SAG)

The SAG shows links between representations of a given entity at different stages of the life cycle. For example, a requirements specification would be linked by an *is_refined_by* link to the relevant design module, and thence by an *is_implemented_by* link to the source code modules which implement it. It also includes *is_tested_by* links between an entity at any life cycle stage and its appropriate test cases.

4.4. Risk interconnection graph (RIG)

The RIG in the AMES RPG is based on Turver's weighted interconnection graph. 'Risk' in this context refers to the effects a modification to one module is likely to have

© 1998 John Wiley & Sons, Ltd.

on other modules. Risk analysis is a useful tool in the process of evaluating alternative solutions in order to select the appropriate maintenance path to follow. The RIG consists of segment and module entities linked by parametrized edges which record the type of the effect a change to one entity might have on another entity, and the probability of that effect occurring. The RIG is populated over a period of time, based on the experience of actual changes made to an application. It is dependent upon an organization recording such change information, which must be manually extracted for inclusion in the RIG.

4.5. Cost interconnection graph (CIG)

The CIG allows a cost to be attached to a proposed change solution. Like the RIG it contains parametrized edges, the parameters recording the cost of changing an entity and all entities affected by that change. Again, such data must be extracted manually from records of actual changes made.

5. IMPACT ANALYSIS USING THE AMES RPG AND IAS

Impact analysis is supported by the AMES Impact Analysis System (IAS). This tool relies on an application being modelled as a set of typed objects connected by a variety of links, e.g., composition links, life cycle traceability links, etc. The technique is completely generic in that it can be applied to any kind of model based on typed objects and links. Thus it can be used to identify potential side-effects at the code level, to determine the impacts of a change at the design level, to identify related documents to be modified, or to determine impacts across life cycle stages, such as to identify the source code modules impacted as a result of a change at the design level.

The approach applies typed modifications to the graph of objects and links, and these modifications are propagated through the graph following previously defined propagation rules. The resulting impacts are themselves modifications, and they are recursively propagated in order to obtain the complete set of impacts.

The IAS thus relies on three generic models. The data model is a representation of an application type, e.g., a C program data model consists of files, functions, *calls* relations, etc., while a requirements document data model would consist of chapters, paragraphs, *refers* relations, etc. The dependency model is an instantiation of the data model, representing the actual components and relations of a particular application. Finally, a propagation model consists of a set of propagation rules which model maintainer knowledge, and it controls when and how a modification is propagated to connected objects.

The AMES RPG provides a data model for the IAS. An instantiation of the RPG provides the dependency model. Given, in addition, a propagation model, the AMES RPG provides the necessary input to the IAS to allow automatic impact analysis.

6. POPULATING THE RPG

The RPG is an object-relationship graph where the objects represent components of a software system, and the links represent various types of dependency relationships between the components. The AU Toolset analyses software system documents to identify various

© 1998 John Wiley & Sons, Ltd.

entities and create links between them. This then provides some of the data required to populate the RPG. This section describes in greater detail the output of the AU Toolset in relation to the requirements for the RPG. Finally, it describes a tool developed to integrate the AU and IAS tools so that the output of the former can be used to generate the dependency file required as input to the IAS.

6.1. HIG

The first task undertaken with the AU Toolset is the creation of hypertext versions of documents. The conversion tools used can be configured to produce an index of 'headings' in a document, e.g., requirement identifiers in a requirements document, or function names in a source code document, with links to their occurrence in the main document. A document can also be split into its constituent parts, thus producing a web of objects (table of contents, sections, etc.) connected by *consists_of* links. This, then, provides the hierarchical structure of a document, which is the content of the hierarchical interconnection graph in the RPG.

6.2. TIG

Linguistic analysis tools within the AU Toolset identify key words in any textual document, including source code, and trace their occurrences through the document. Code analysis tools are used to produce call graphs and control flow graphs from source code (at present only C source can be analysed). One of the document analysis tools generates links from function names in source code to references to those functions in other documents. The output of these various tools, then, is semantic links which can be used to populate the thematic interconnection graph. At present there is no tool support for the automatic generation of thematic data from design documents.

6.3. SAG

Life cycle analysis tools use the output of linguistic analysis tools to create links between words or phrases occurring in documents at different life cycle stages. Thus, links are created between a requirements identifier and the relevant design modules, and from the design module to source code modules which implement it, such links being the content of the SAG. (These tools are currently specific to ESA standard identifiers.)

6.4. RIG and CIG

The data for population of these graphs must be manually extracted from previous release information and the graphs manually annotated.

6.5. AU/IAS integration

In order to demonstrate the principle of integration of application understanding and impact analysis, a tool has been developed which utilizes the output of the life cycle

© 1998 John Wiley & Sons, Ltd.

analysis tools to populate the SAG. It takes one of the files generated by the life cycle analysis tools, extracts the required dependency data and writes it out to a dependency file in the format required by the IAS. The tool has been successfully used on documentation from one of the AMES case studies to allow high level impact analysis to be undertaken.

7. CONCLUSIONS AND FURTHER WORK

This paper has described two areas of work from the AMES project, dedicated to supporting the work of the maintenance engineer in determining the full effect of a proposed change on a software system. the AU Toolset presents textual and graphical views of a system in a format which aids the process of understanding the functionality of a system. Both the output of the AU Toolset, and the understanding gained by the maintainer, can be used in the population of a ripple propagation graph. The RPG provides several views of a system, which can be automatically analysed to identify all components which may be affected by a proposed change.

The AU Toolset is currently a prototype, developed to analyse a specified range of documentation as used by AMES project partners. Some of the tools require further work to enable them to analyse documentation using different formats. In addition, further tools are required to enable population of those areas of the RPG not at present supported. It has been used successfully in a number of case studies, both internally and commercially by other partners in the AMES project. It has been successfully demonstrated that the output of the AU Toolset can be used via an integration tool to generate automatically the dependency files to populate the SAG element within the RPG. A further case study is currently being undertaken on a large COBOL application.

The IAS tool has also been prototyped and used in a number of studies.

The specification of the RPG provides an important landmark in the identification of the kinds of information generated by the AU Toolset which are relevant to supporting high level impact analysis. The groundwork is now in place for the integration of the AMES AU and IA tools to provide a comprehensive toolset for thorough impact analysis of a proposed change early within the maintenance process.

Acknowledgements

The work described in this paper has been carried out with the support of the European Union under the ESPRIT programme, Project 8156, AMES. Our collaborators in this project have been as follows: Cap Gemini Innovation, Cap Programmator, Intecs Sistemi Spa, Matra Marconi Space, OPL-TT, Space Systems Finland and Valtion Teknillinen Tutkimuskeskus (VTT). In addition to the authors, individuals from the above organizations have contributed to the work reported here.

References

Agusa, K., Kishimoto, Y. and Ohno, Y. (1983) ‘A supporting system for software maintenance’, in Teichroew, G. and David, G. (Eds), *System Description Methodologies*, Proceedings of IFIP TC2, North Holland, Amsterdam, pp. 481–503.

Arthur, L. J. (1988) *Software Evolution: The Software Maintenance Challenge*, John Wiley & Sons, New York, NY, 254 pp.

© 1998 John Wiley & Sons, Ltd.

Barros, S., Bodhuin, Th., Escudié, A., Queille, J. P. and Voidrot, J. F. (1995) 'Supporting impact analysis: a semi-automated technique and associated tool', *Proceedings of the International Conference on Software Maintenance ICSM 95*, IEEE Computer Society Press, Los Alamitos CA. pp. 42–51.

Battaglia, M. and Savoia, G. (1995) 'Reverse—NICE: a re-engineering methodology and supporting tool', *Proceedings of the Second International Eurospace—Ada-Europe Symposium*, Lecture Notes in Computer Science Series, Volume 1031, Springer–Verlag, Berlin, pp. 244–248.

Boldyreff, C., Burd, E. and Hather, R. (1994) 'An evaluation of the state of the art for application management', *Proceedings of the International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 161–169.

Boldyreff, C., Burd, E., Hather, R., Mortimer, R., Munro, M. and Younger, E. (1995) 'The AMES approach to application understanding: a case study', *Proceedings of the International Conference on Software Maintenance ICSM 95*, IEEE Computer Society Press, Los Alamitos CA, pp. 182–191.

Brooks, R. (1983) 'Towards a theory of comprehension of computer programs', *IEEE Computer*, **16**(10), 18–37.

Burd, E. L., Younger, E. J. and Boldyreff, C. (1994) 'Improving program comprehension through the application of learning theory', AMES Project Working Paper in University of Durham Computer Science Technical Report 2/94; *Collected Papers of the AMES Project*, University of Durham, Durham, U.K., Volume 1, Paper 5, pp. 1–15.

Fillon, P. (1994) 'An approach to impact analysis in software maintenance', M.Sc. Thesis, Department of Computer Science, University of Durham, Durham, U.K., 108 pp.

Fyson, M. J. (1995) 'An investigation into methods of improving program comprehension through better documentation', Final Year Project Report, Department of Computer Science, University of Durham, Durham, U.K., 79 pp.

Gagne, R. M. (1974) *Essentials of Learning for Instruction*, Dryden Press, London, 164 pp.

Gilmore, D. J. (1991) 'Models of debugging', *Acta Psychologia*, **78**(1), 151–172.

Laitinen, K. and Mukari, T. (1992) 'Disciplined natural naming', *Proceedings of the 25th Hawaii International Conference on System Sciences, Vol. II: Software Technology*, IEEE Computer Society Press, Los Alamitos CA, pp. 91–100.

Laitinen, K. (1995) 'Enhancing maintainability of source programs through disabbreviation', *Journal of Systems and Software*, **37**(2), 117–128.

Letovsky, S. (1986) 'Cognitive processes in program comprehension', in Soloway, E. and Lyengar, S. (Eds), *Empirical Studies of Programmers*, Ablex, Norwood NJ, pp. 58–79.

Lientz, B., Swanson, E. B. and Tompkins, G. E. (1978) 'Characteristics of application software maintenance', *Communications of the ACM*, **21**(6), 466–471.

Livadas, P. E. and Alden, S. D. (1993) 'A toolset for program understanding', *IEEE Second Workshop on Program Comprehension WPC'93*, IEEE Computer Society Press, Los Alamitos CA, pp. 110, 118.

Pressman, R. S. (1992) *Software Engineering: A Practitioner's Approach*, 3rd Edition, McGraw-Hill, Inc., New York NY, 793 pp.

Standish, T. A. (1984) 'An essay on software reuse', *IEEE Transactions on Software Engineering*, **SE-10**(5), 494–497.

Swanson, E. B. (1976) 'The dimension of maintenance', *Proceedings 2nd International Conference of Software Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 492–497.

Turver, R. J. (1993) 'Early detection of ripple propagation in evolving software systems', Ph.D. Thesis, Department of Computer Science, University of Durham, Durham, U.K., 273 pp.

Younger, E. J. and Bennett, K. H. (1993) 'Model based tools to record program understanding', *2nd Workshop on Program Comprehension, WPC'93*, IEEE Computer Society Press, Los Alamitos CA, pp. 87–95.

© 1998 John Wiley & Sons, Ltd.

Authors' biographies:

M. Joanna Fyson graduated with a First Class Honours B.Sc. degree in Computer Science from the University of Durham in 1995. From August 1995 to July 1996, she has worked as a research assistant on the AMES project at the University of Durham. She is now employed by Sainsbury's plc working in IT management.

Cornelia Boldyreff has been a lecturer in the Department of Computer Science at the University of Durham since April 1992. She led the AMES project wok at Durham from its start in 1993 to the project's successful conclusion in July 1996. E-mail: Cornelia.Boldyreff@durham.ac.uk

© 1998 John Wiley & Sons, Ltd.